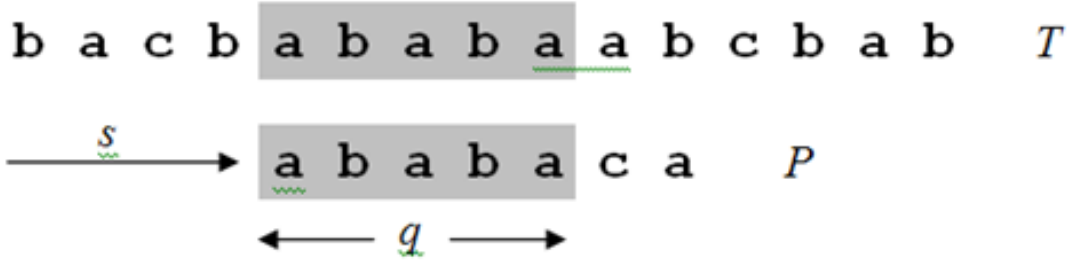
Курс «Алгоритмы и алгоритмические языки»
1 семестр 2018/2019

Лекция 17

Алгоритм Кнута – Морриса – Пратта. Идея

- *Префикс-функция*, ассоциированная с образцом *P*, показывает, где в строке *P* повторно встречаются различные префиксы этой строки. Если это известно, можно не проверять заведомо недопустимые сдвиги.
- ***Пример***. Пусть ищутся вхождения образца *P* = **a b a b a c a** в текст *T*. Пусть для некоторого сдвига *s* оказалось, что первые *q* символов образца совпадают с символами текста. Значит, символы текста от *T*[*s*+1] до *T*[*s*+*q*] известны, что позволяет заключить, что некоторые сдвиги заведомо недопустимы.



Алгоритм Кнута – Морриса – Пратта. Идея

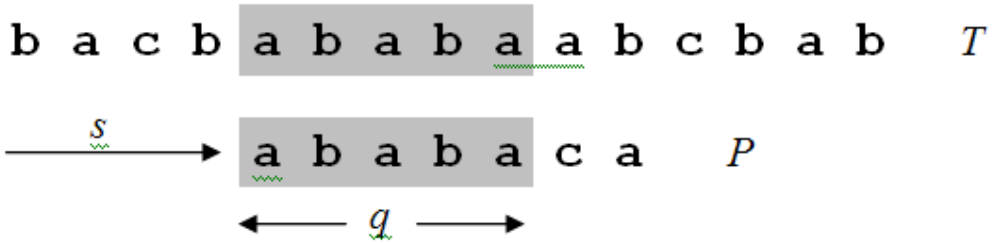
- Пусть $P[1..q] = T[s+1..s+q]$; каково минимальное значение сдвига $s' > s$, для которого $P[1..k] = T[s'+1..s'+k]$, где $s'+k = s+q$?
  - Число s' - минимальное значение сдвига, большего s, которое совместимо с тем, что $T[s+1..s+q] = P[1..q]$. Следовательно, значения сдвигов, меньшие s', проверять не нужно.
    Лучше всего, когда $s' = s+q$, так как в этом случае не нужно рассматривать сдвиги $s+q-1, s+q-2, \ldots, s+1$.
    Кроме того, при проверке нового сдвига s' можно не рассматривать первые его k символов образца: они заведомо совпадут.
- Чтобы найти s', достаточно знать образец P и число q:
  $T[s'+1..s'+k]$ – суффикс P_q, поэтому k – это наибольшее число, для которого P_k является суффиксом P_q. Зная k (число символов, заведомо совпадающих при проверке нового сдвига s'), можно вычислить s' по формуле $s' = s + (q - k)$.

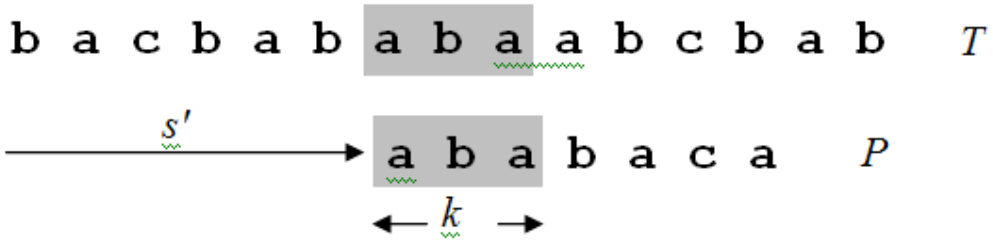
Алгоритм Кнута – Морриса – Пратта. Префикс-функция

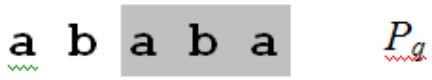
- ***Определение***. Префикс-функцией, ассоциированной со строкой $P[1..m]$, называется функция π: $\{1,2, \ldots, m\} \to \{0,1, \ldots, m-1\}$, определенная следующим образом:

$$\pi[q] = \max\{k: k<q \ \& \ P_k \succ P_q\}$$

Иными словами, $\pi[q]$ – длина наибольшего префикса P, являющегося суффиксом P_q.







Алгоритм Кнута – Морриса – Пратта. Префикс-функция

```
void prefix_func (char *pat, int *pi, int m) {
  int k, q;

  /* Считаем, что pat и pi нумеруются от 1 */
  pi[1] = 0; k = 0;
  for (q = 2; q <= m; q++) {
    while (k > 0 && pat[k + 1] != pat[q])
      k = pi[k];
    if (pat[k + 1] == pat[q])
      k++;
    pi[q] = k;
  }
}
```

Алгоритм Кнута – Морриса – Пратта. Префикс-функция

- ***Лемма 1***. Обозначим $\pi^*[q] = \{q, \pi[q], \pi^2[q], ..., \pi^t[q]\}$,
  где $\pi^i[q]$ есть i-я итерация префикс-функции, $\pi^t[q] = 0$.
  Пусть *P* – строка длины *m* с префикс-функцией π.
  Тогда для всех *q* = 1, 2, ..., *m* имеем $\pi^*[q] = \{k : P_k \succ P_q\}$.
- Лемма показывает, что при помощи итерирования префикс-функции можно для данного *q* найти все такие *k*, что P_k является суффиксом P_q.
- Доказательство.

  (1) Покажем, что если *i* принадлежит $\pi^*[q]$, то P_i является суффиксом P_q.

  Действительно, $P_{\pi[i]} \succ P_i$ по определению префикс-функции, так что каждый следующий член последовательности $P_i, P_{\pi[i]}, P_{\pi[\pi[i]]}, ...$ является суффиксом всех предыдущих.

Алгоритм Кнута – Морриса – Пратта. Префикс-функция

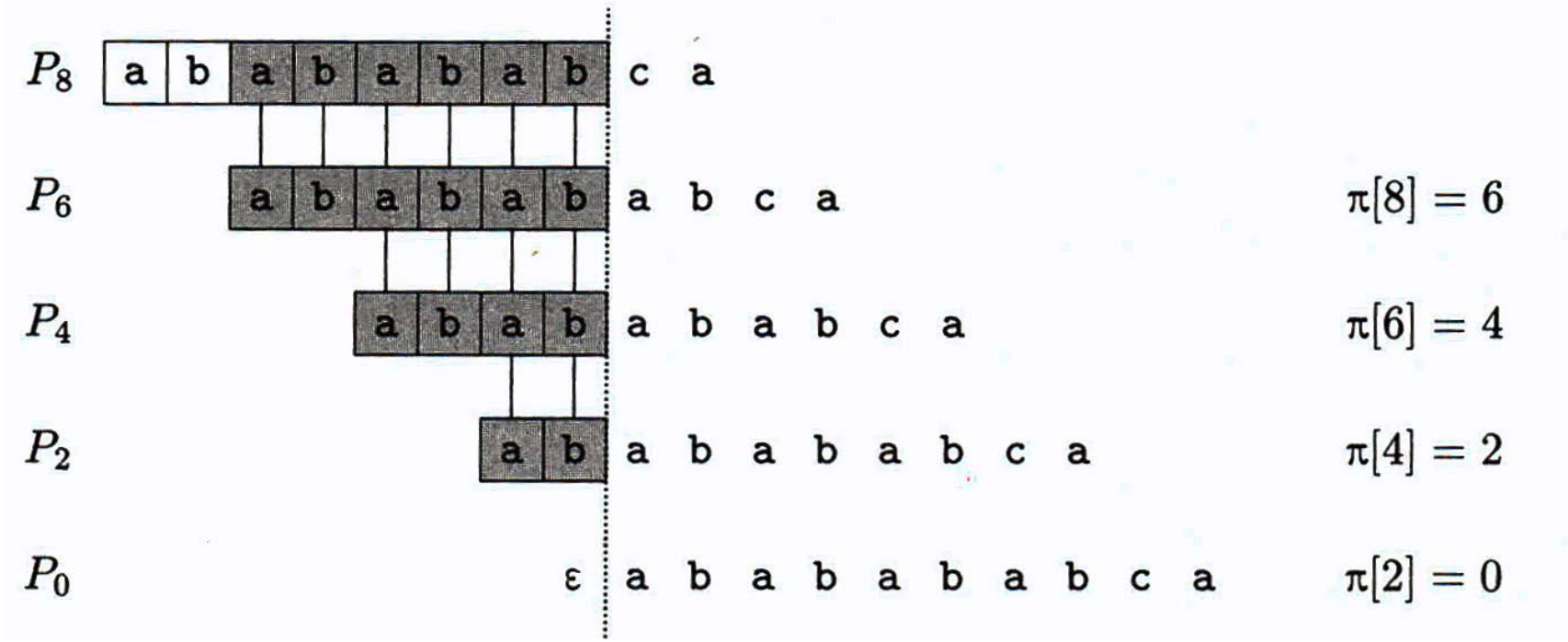
- Доказательство.

  (2) Покажем, что наоборот, если P_i является суффиксом P_q, то i принадлежит $\pi^*[q]$.

  - Расположим все P_i , являющиеся суффиксами P_q, в порядке уменьшения i (длины): P_{i_1}, P_{i_2},…
  - Покажем по индукции, что $P_{i_k} = \pi^k[q]$.
  - База индукции (k=1): для максимального префикса P_i , являющегося суффиксом P_q, по определению $i=\pi[q]$.
  - Шаг индукции: если $P_{i_k} = \pi^k[q]$, то по определению $j = \pi[\pi^k[q]]$ соответствует максимальный префикс P_j, который является суффиксом P_{i_k}. Обе строки P_j и P_{i_k} есть суффиксы P_q по построению. Таким максимальным префиксом из оставшихся $P_{i_{k+1}}$, $P_{i_{k+2}}$,… по построению является префикс $P_{i_{k+1}}$,то есть $j = i_{k+1}$.
  - (2) можно доказать и от противного: для наибольшего числа j такого, что $P_j \succ P_q$, но j не входит в $\pi^*[q]$, определение префикс-функции нарушается.

Алгоритм Кнута – Морриса – Пратта. Префикс-функция

$P[i]$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1



$$\pi^*[8] = \{8,6,4,2,0\}$$

Алгоритм Кнута – Морриса – Пратта. Префикс-функция

- ***Лемма 2***. Пусть *P* – строка длины *m* с префикс-функцией π. Тогда для всех *q* = 1, 2, ..., *m*, для которых $\pi[q] > 0$, имеем $\pi[q]-1 \in \pi^*[q-1]$.
- Доказательство.
  - Если $k = \pi[q] > 0$, то P_k является суффиксом P_q по определению префикс-функции.
  - Следовательно, P_{k-1} является суффиксом P_{q-1}.
  - Тогда по Лемме 1 $k-1 \in \pi^*[q-1]$, т.е. $\pi[q]-1 \in \pi^*[q-1]$.
- Определим множества E_{q-1} как

  $E_{q-1} = \{k : k \in \pi^*[q-1] \text{ и } P[k+1] = P[q]\}$.

  Множество E_{q-1} состоит из таких *k*, что P_k является суффиксом P_{q-1}, и за ними идут одинаковые буквы *P*[*k*+1] и *P*[*q*].

  Из определения вытекает, что P_{k+1} есть суффикс P_q.

Алгоритм Кнута – Морриса – Пратта. Префикс-функция

- ***Следствие 1***. Пусть P – строка длины m с префикс-функцией π. Тогда для всех $q = 2, 3, ..., m$

$$\pi[q] = \begin{cases} 0, \text{если } E_{q-1} \text{ пусто;} \\ 1 + \max\{k \in E_{q-1}\}, \text{если } E_{q-1} \text{ не пусто.} \end{cases}$$

- Доказательство.
  - Если $r = \pi[q] \geq 1$, то $P[r] = P[q]$ и по Лемме 2 $r - 1 = \pi[q] - 1 \in \pi^*[q-1]$.
  - Т.к. $P[r] = P[q]$, то $P[(r\text{-}1)+1] = P[q]$.
  - Поэтому $r - 1 \in E_{q-1}$ по определению E_{q-1} и из $\pi[q] \geq 1$ следует непустота E_{q-1}.
  - Следовательно, если E_{q-1} пусто, то $\pi[q] = 0$ (от противного).
  - Если $k \in E_{q-1}$, то P_{k+1} есть суффикс P_q (из определения), следовательно, $\pi[q] \geq k + 1$ и $\pi[q] \geq 1 + \max\{k \in E_{q-1}\}$.
  - То есть, если E_{q-1} не пусто, то префикс-функция положительна. Но тогда $\pi[q] - 1 \in E_{q-1}$, $\pi[q] - 1$ не больше максимума из E_{q-1}, т.е. $\pi[q] \leq 1 + \max\{k \in E_{q-1}\}$.

Алгоритм Кнута – Морриса – Пратта. Префикс-функция

```
1  void prefix_func (char *pat, int *pi, int m) {
2    int k, q;
3
4    /* Считаем, что pat и pi нумеруются от 1 */
5    pi[1] = 0; k = 0;
6    for (q = 2; q <= m; q++) {
7      while (k > 0 && pat[k + 1] != pat[q])
8        k = pi[k];
9      if (pat[k + 1] == pat[q])
10       k++;
11     pi[q] = k;
12   }
13 }
```

Алгоритм Кнута – Морриса – Пратта. Префикс-функция

- ***Теорема 1***. Функция **`prefix_func`** правильно вычисляет префикс-функцию π.

- Доказательство.
  - Покажем, что при входе в цикл функции $k = \pi[q\text{-}1]$.
  - <u>База индукции</u>.

    При **`q = 2`** **`k = 0`**, **`pi[q-1] = pi[1] = 0`**.
  - <u>Шаг индукции</u>.

    Пусть при входе в цикл функции $k = \pi[q\text{-}1]$.

    Код на строках 7-8

```
while (k > 0 && pat[k + 1] != pat[q])
  k = pi[k];
```

    находит наибольший элемент E_{q-1} (т.к. цикл перебирает в порядке убывания элементы из $\pi^*[q\text{-}1]$ и для каждого проверяет равенство **`pat[k + 1] != pat[q])`**).

Алгоритм Кнута – Морриса – Пратта. Префикс-функция

- ***Теорема 1***. Функция **`prefix_func`** правильно вычисляет префикс-функцию π.

- Доказательство.
  - После выхода из цикла на строках 7-8

```
while (k > 0 && pat[k + 1] != pat[q])
  k = pi[k];
```

1) если **`pat[k + 1] == pat[q]`**, то выполняется код на строке 10:

```
k++;
```

что из Следствия 1 дает нам $\pi[q]$.

2) если **`pat[k + 1] != pat[q]`**, то **`k == 0`**, множество E_{q-1} пусто и $\pi[q] = 0$.

Алгоритм Кнута – Морриса – Пратта. Функция kmp

```
void kmp (char *text, char *pat, int m, int n) {
  int q;
  int pi[m + 1];  /* VLA-массив */
  /* Через alloca: int *pi = alloca ((m + 1) * sizeof (int)); */

  /* Считаем, что pat и text нумеруются от 1 */
  prefix_func (pat, pi, m);
  q = 0;
  for (i = 1; i <= n; i++) {
    while (q > 0 && pat[q + 1] != text[i])
      q = pi[q];
    if (pat[q + 1] == text[i])
      q++;
    if (q == m) {
      printf ("образец входит со сдвигом %d\n", i - m);
      q = pi[q];
  }
}
```

Алгоритм Кнута – Морриса – Пратта. Функция `kmp`

- Алгоритм КМП для подстроки *P* и текста *T* эквивалентен вычислению префикс-функции для строки *Q* = *P*#*T*, где # – символ, заведомо не встречающийся в обеих строках
  - Длина максимального префикса *Q*, являющегося её суффиксом (т.е. значение префикс-функции), не превосходит длины *P*
  - Допустимый сдвиг обнаруживается в тот момент, когда очередное вычисленное значение префикс-функции совпадает с длиной подстроки *P* (условие **`if (q == m)`**)
  - В явном виде объединенная строка не строится!
- ***Теорема 2***. Функция **`kmp`** работает правильно.
  - Формальное доказательство осуществляется по аналогии с доказательством Теоремы 1, где множества, подобные E_{q-1}, строятся для строки-текста, а не строки-образца.
- Свойства префикс-функции часто используются и в других задачах (кроме поиска подстроки в строке)
  - Полезной оказывается Лемма 1: итерированием префикс-функции можно найти все префиксы строки, являющиеся ее суффиксами

Алгоритм Кнута – Морриса – Пратта. Время работы

- Функция **prefix_func** выполняет $\le (m - 1)$ итераций цикла **`for`**. Стоимость каждой итерации можно считать равной $O(1)$, а стоимость всей процедуры $O(m)$.
  - Каждая итерация цикла **`while`** (строки 7-8) уменьшает **k**
  - Увеличивается **k** только в строке 10 не более одного раза на итерацию цикла **`for`** (строки 6-11)
  - Следовательно, операций уменьшения не больше, чем итераций цикла **`for`**, то есть $\le (m - 1)$ на весь цикл и $O(1)$ на итерацию в среднем
- Аналогично, функция **kmp** выполняет $\le (n - 1)$ итераций, и ее стоимость (без учета вызова **prefix_func**) есть $O(n)$. Следовательно, время выполнения всей процедуры $O(m + n)$.

Динамические структуры данных. Стек

- **Стек** (*stack*) – это динамическая последовательность *элементов*, количество которых изменяется, причем как добавление, так и удаление элементов возможно только с одной стороны последовательности (вершина стека).
- Работа со стеком осуществляется с помощью функций:

  `push(x)` – *затолкать* элемент **`x`** в стек;

  `x = pop()` – *вытолкнуть* элемент из стека.
- Стек можно организовать на базе:
  - фиксированного массива **`stack[MAX]`**, где константа **`MAX`** задает максимальную глубину стека.
  - динамического массива, текущий размер которого хранится отдельно.
  - в обоих случаях необходимо хранить позицию текущей вершины стека.
  - можно использовать и другие структуры данных (например, список).

Организация стека на динамическом массиве

```
struct stack {
  int sp;   /* Текущая вершина стека */
  int sz;   /* Размер массива */
  char *stack;
} stack = { .sp = -1, .sz = 0, .stack = NULL };

static void push (char c) {
  if (stack.sz == stack.sp + 1) {
    stack.sz = 2*stack.sz + 1;
    stack.stack = (char *) realloc (stack.stack,
                  stack.sz*sizeof (char));
  }
  stack.stack[++stack.sp] = c;
}
```

Организация стека на динамическом массиве

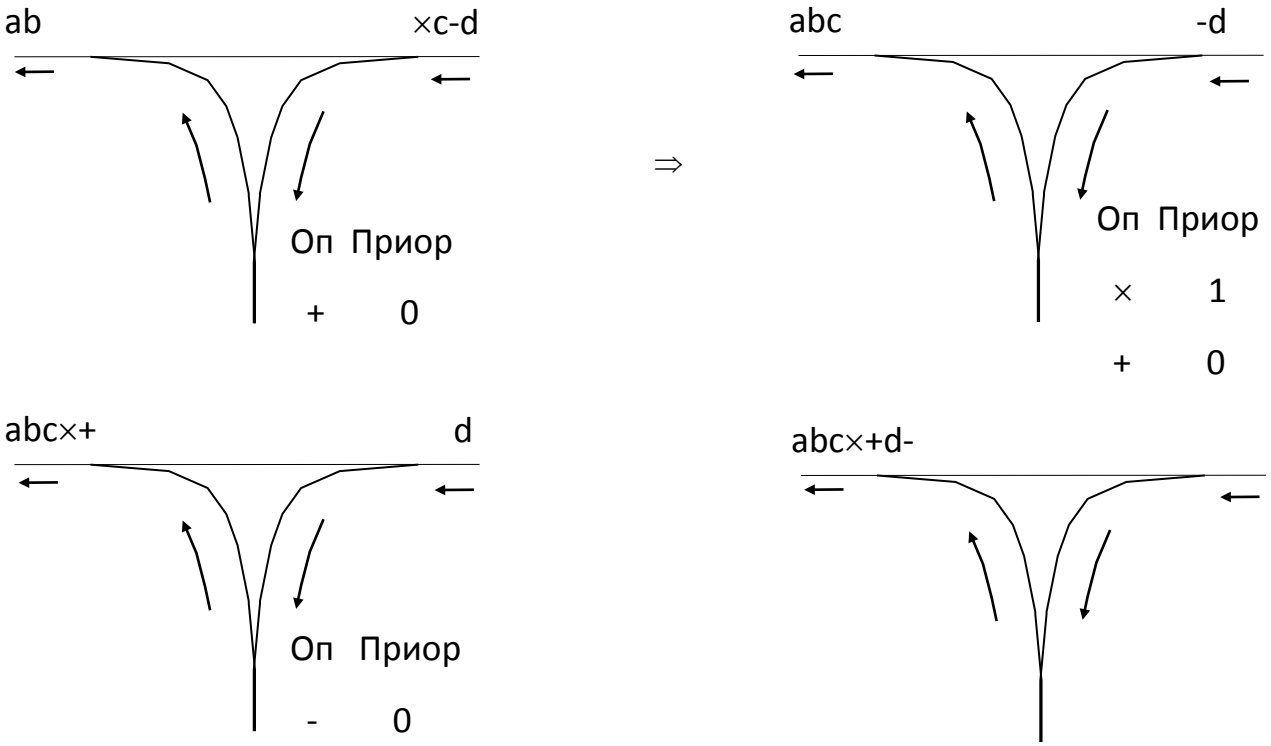
```
struct stack {
  int sp;   /* Текущая вершина стека */
  int sz;   /* Размер массива */
  char *stack;
} stack = { .sp = -1, .sz = 0, .stack = NULL };

static char pop (void) {
  if (stack.sp < 0) {
    fprintf (stderr, "Cannot pop: stack is empty\n");
    return 0;
  }
  return stack.stack[stack.sp--];
} // Дома. Сделайте, чтобы результат записывался по указателю-аргументу,
а функция возвращала код успеха операции.

static int isempty (void) {
  return stack.sp == -1;
}
```

Пример работы со стеком

- Перевод арифметического выражения в обратную польскую запись (постфиксную).

a + b × c - d → abc × + d -

c × (a + b) - (d + e)/f → cab + × de + f/-



Пример работы со стеком

- Перевод арифметического выражения в обратную польскую запись.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

#include "stack.c"

/* Считывание символа-операции или переменной */
static char getop (void) {
  int c;
  while ((c = getchar ()) != EOF && isblank (c))
    ;
  return c == EOF || c == '\n' ? 0 : c;
}
```

Пример работы со стеком

- Перевод арифметического выражения в обратную польскую запись.

```
/* Является ли символ операцией */
static int isop (char c) {
  return (c == '+') || (c == '-') || (c == '*')
         || (c == '/');
}
/* Каков приоритет символа-операции */
static int prio (char c) {
  if (c == '(')
    return 0;
  if (c == '+' || c == '-')
    return 1;
  if (c == '*' || c == '/')
    return 2;
  return -1;
}
```

Пример работы со стеком

- Перевод арифметического выражения в обратную польскую запись.

```
int main (void) {
  char c, op;

  while (c = getop ()) {
    /* Переменная-буква выводится сразу */
    if (isalpha (c))
      putchar (c);
    /* Скобка заносится в стек операций */
    else if (c == '(')
      push (c);
    else <...>
```

Пример работы со стеком

- Перевод арифметического выражения в обратную польскую запись.

```
/* Операция заносится в стек в зависимости от приоритета */
else if (isop (c)) {
       while (! isempty ()) {
         op = pop ();
         /* Заносим, если больший приоритет */
         if (prio (c) > prio (op)) {
           push (op); break;
         } else
         /* Иначе выталкиваем операцию из стека */
         putchar (op);
       }
       push (c);
    } else <...>
```

Пример работы со стеком

- Перевод арифметического выражения в обратную польскую запись.

```
/* Скобка выталкивает операции до парной скобки */
} else if (c == ')')
      while ((op = pop ()) != '(')
        putchar (op);
  }
/* Вывод остатка операций из стека */
  while (! isempty ())
    putchar (pop ());
  putchar ('\n');
  return 0;
}
```

Дома. Введите операцию peek() и перепишите код с ее помощью. Обработайте случай непарных скобок.